

# Command Processing

Gordon Watts  
6-26-97  
Online Meeting

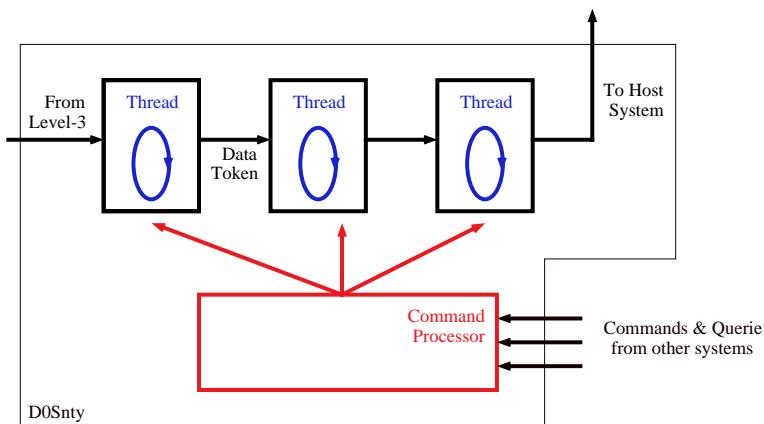
- Command Processing

You can find all of this talk on WWW:  
<http://d0sgi0.fnal.gov/gwatts/talks/>

# Command Classes

---

First a little context (DØSNTY):



- Many threaded application
- Commands must interact with threads
- Commands aren't part of threads

# Command Types

---

- Slow Commands

- Wait for program to complete operation before disconnecting
- Sometimes very long operations (seconds or minutes?).
- **Example:** Reset/Restart, Empty Queues, etc.

- Quick Commands

- Never requires interacting or blocking another thread
- Start a operation but don't wait for its completion
- **Examples:** Quit as soon as empty, How many events processed?

Slow Commands should not block  
Quick Commands

# Other Requirements

---

- Command library is independent of I/O method
  - TCP/IP, DØIP, Named Pipes, etc.
- Independent of format of input data stream
  - text or binary
- Auto processing of incoming commands
- Commands can have conversations (context)
- Easy to use (**Yeah, Right!**).

# Added Requirements

---

Last presentation added a few things:

- **More than one** command per conversation
- **Batched sets of commands**
- **Don't force one** to use all the functionality
  - Can just use the command\_processor and write a command processing loop.

This has caused a shift:

- Centered around a **command queue**
- **Push functionality down to transport layer**
- **Keep threads few**

# The Command Processor

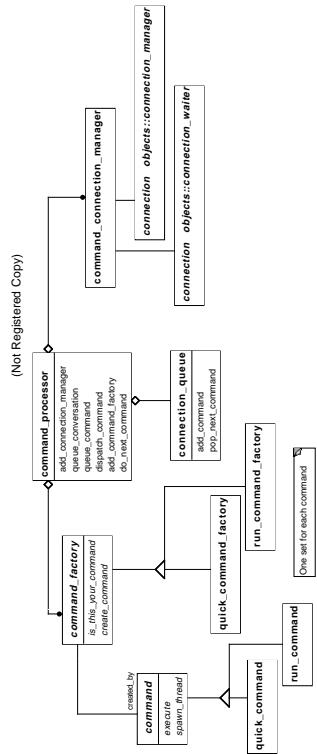
---

Everything is based around the `command_processor` class:

- Has a `connection queue`.
- Maintains `two threads` per connection
  1. Waits for new connections
  2. Group wait for inputs on old connections
- Command class access will be `split off`

# The Command Processor

---



# Command Dispatch

---

Command dispatch is done by a subclass of the `command_processor` class.

- Maintains a list of `command_factory` classes
  - One `command_factory` per command
- Methods to manage list (add/remove commands).
- Knows about a `connection_manager`

# Connection Classes

---

The `message` is the abstract object passed around

- Fed to and gotten from a `connection_conversation`
- Can block waiting for a new message

The `connection_manager` keeps track of connections.

- Waits for new connection to a *port*
- Blocks threads

# Context Object

---

The `conversation_context` object:

- Maintains a “context” for a conversation.
- Owns a list of `context_item` objects.
- User creates objects that inherit from `context_item`.
- When the conversation is deleted, so are the `context_items`.

**For example:** Blocking other programs from configuring subdetectors.

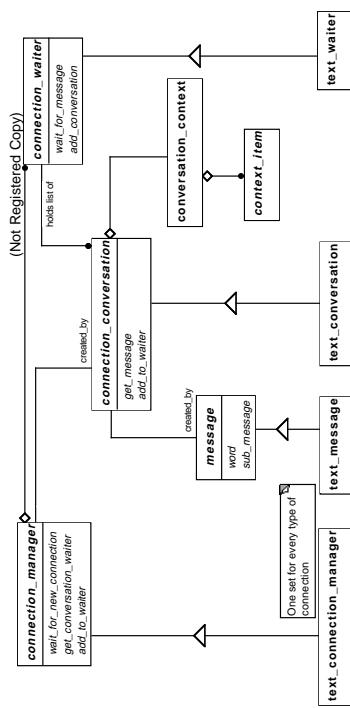
# Waiters

---

`connection_waiter` waits for input on a collection of conversations

- The `connection_manager` knows what waiters exist.
- The waiters are `referenced by proxy`
- Waiter will `block a thread` till a conversation has a `complete` message.
- `This is the hard part about implementing a new communications protocol.`

# Connection Classes



# Creating

---

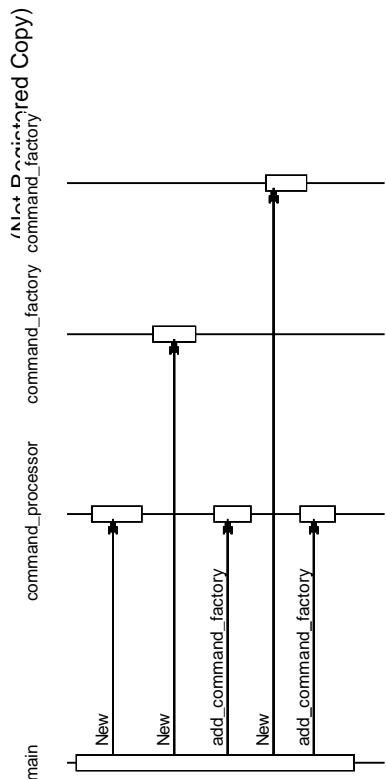
Create a new command class.

- `get_next_ready_conversation` method
  - Returns next ready conversation for reading
  - Method will block till conversation ready
  - Call in a main loop.
  - If you want to write command loop.
- Subclass `dispatch_command`
  - Called when a new command is available.
- Add `command` classes
  - As before.

# Creating

---

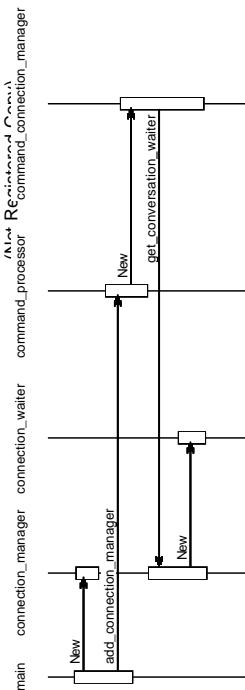
No code yet, no time.



# Add a connection manager

---

- Simple for you.
- Do it any time
- Lots of work for library.



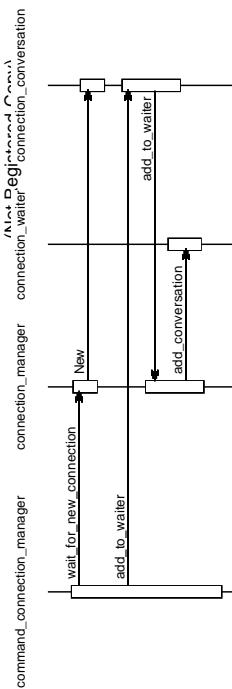
# Conclusions

---

- There are more event traces on the web.
- Details of internal receive message, etc. left out.
- No code yet (except last version).

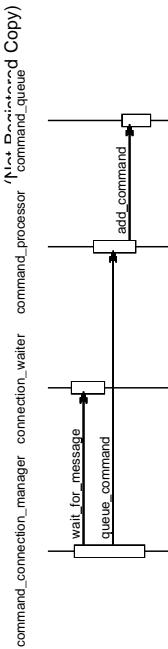
# New Conversation

---



# Waiting for Message

---



# Threading still same

---

